Abstract:

In the age of algorithmically mediated information, public perceptions of nutrition are increasingly shaped by unverified digital narratives. This application aims to build a chatbot from scratch that delivers reliable information on personal nutritional requirements based on the individual profile features such as age, BMI, daily activity and medical conditions. It implements natural language Processing techniques to train a bigram and transformer-based language models on the OpenWebText corpora dataset, and the textual data from 57 government-issued and academic nutrition PDF documents. By training a Transformer-based language model on both corpora, the developed chatbot is intended to provide nutritional recommendations tailored to a person's specific characteristics based on individual queries. This presents a valuable opportunity in creating a global impact by utilizing large volume of existing text-based academic and governmental research documents to provide reliable nutritional guidance for the public, especially in developing countries where nutritional deficiency has been a major issue.

Table of contents

Introduction

"The limits of my language mean the limits of my world."
—Ludwig Wittgenstein

Language model pretraining has shown to be effective in various natural language processing tasks from natural language inference, question answering to paraphrasing where the relationship between sentences or words are predicted holistically as well as on the token level (Radford et al., 2018). Language models can do this by modeling the generative likelihood of word sequences by predicting the probabilities of the new missing tokens (Zhao et al., 2025). Language modeling first began during the 1990s as statistical language models which used statistical learning methods to predict the next words based on the most recent context (Rosenfeld, 2000). One example of such a model is the bigram language model, which is computationally lightweight and implemented in our application. However, it suffers significantly from the curse of dimensionality and produced completely irrelevant results in our context (Zhao et al., 2025).

Neutral Language models introduced in 2013, which predicts the probability of word sequences using neural networks such as multi-layer perceptron and recurrent neutral networks was a significant milestone in the history of language modeling for NLP tasks (Zhao et al., 2025). Moreover, the introduction of word2vec which proved to be highly effective not just to predict the next word in a sentence, but to learn deeper representations of language which significantly influenced the development of modern NLP. It was possible due to its ability to map word embeddings in a continuous vector space (Mikolov et al., 2013). This was followed by the pretrained model such as ELMO which was able to capture context within the word representations utilizing the LSTM architecture which its predecessors were unable to do.

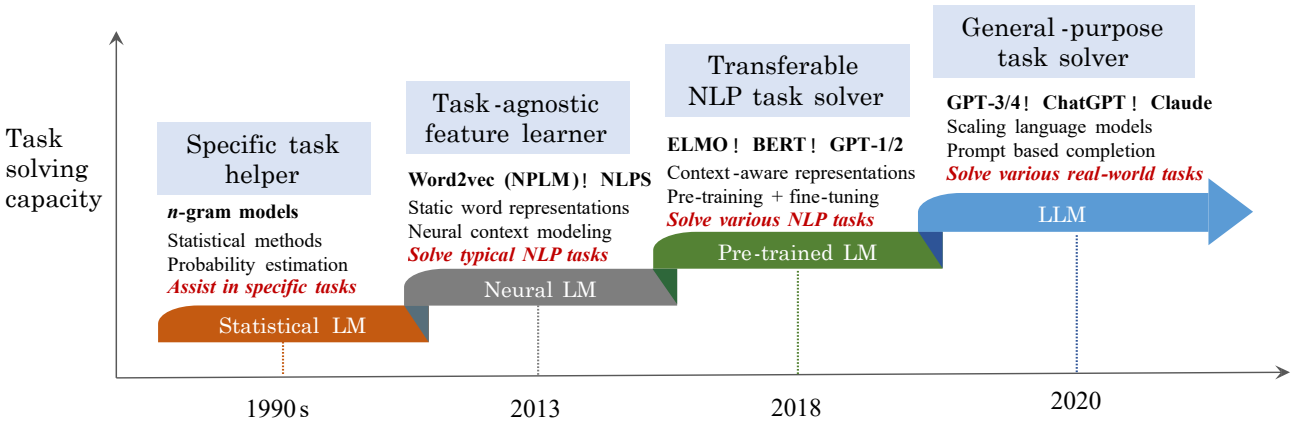


Fig: Evolution of the language models

These breakthroughs, however, pales in comparison to the Transformer architecture developed by researchers at Google Brain, which many AI professionals consider as the most significant paper in the

field of modern AI (Vaswani et al., 2017). The transformer architecture, which is highly parallelizable, was able to significantly increase the performance of pretrained language models introducing BERT and GPT-1/2 (Devlin et al., 2019; Radford et al., 2018). Because of the robust architecture of the transformer which is highly scalable and parallelizable and able to capture rich relationships between distant words through self-attention mechanism, almost all the advanced chatbots available today are built on this architecture.

One of the recent breakthroughs in language model research was achieved through an innovative modification of the attention mechanism by DeepSeek (2024). The Key and Value components are essential in the attention mechanism of the transformer architecture, allowing language models to capture relationships across tokens. Traditionally, these values are stored in a cache during inference, leading to substantial memory usage. DeepSeek's development of Multi-Head Latent Attention (MLA) introduced a method to compress the Key and Value representations for each token into a single latent vector. This approach significantly reduces the KV cache size by reconstructing the full Key and Value vectors on-the-fly during inference, thereby enhancing memory efficiency without compromising performance.

Architectural comparison between traditional language models and DeepSeek

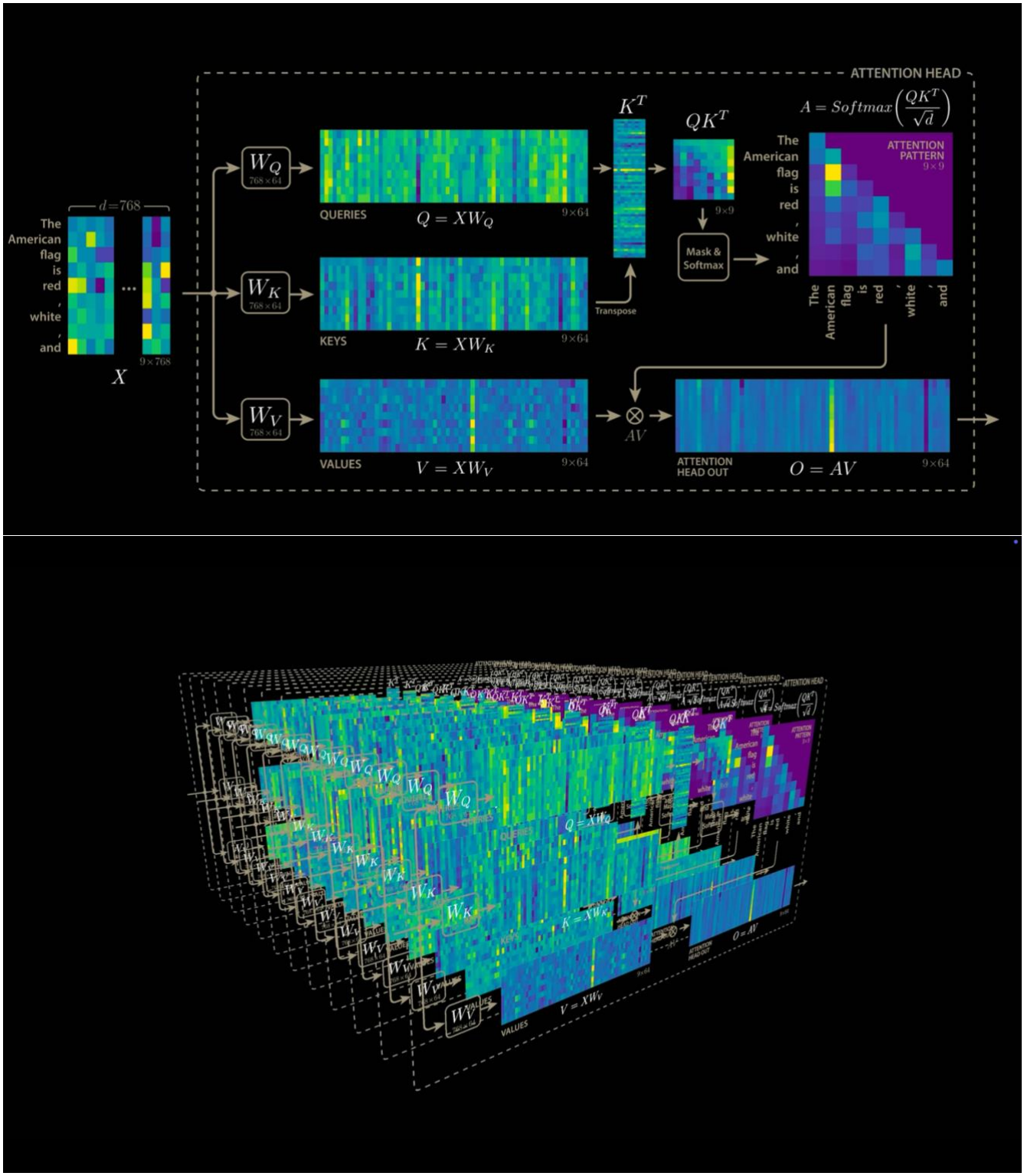


Fig: Multi Attention heads of traditional language models also used in this application

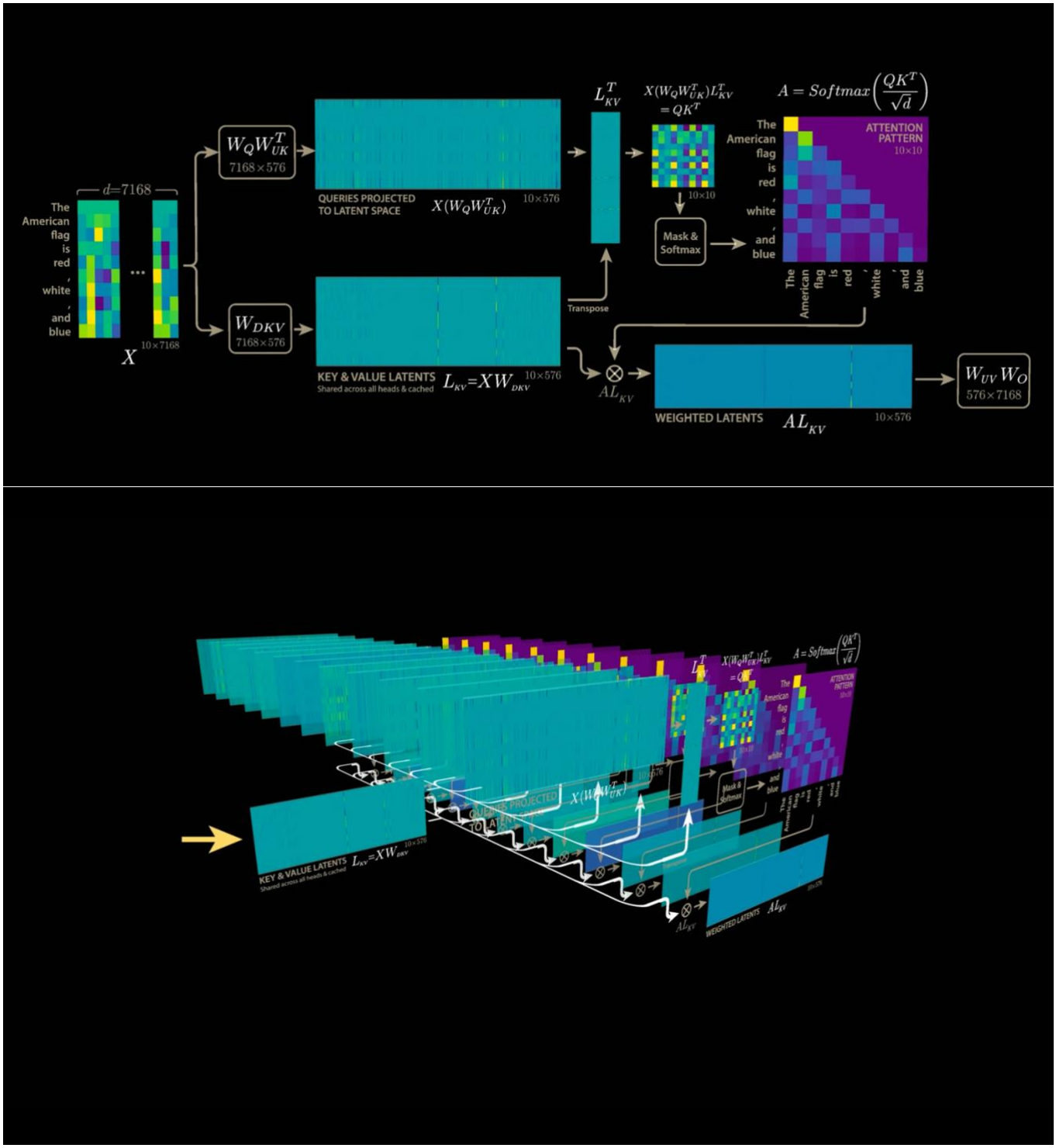


Fig: Multi-Head Latent Attention introduced by DeepSeek

Although natural language processing techniques have been advancing exponentially, studies on implementing such language models for nutritional recommendation systems are still lacking. Nutrition deficiency refers to significantly lowered levels of one or more nutrients, making the body incapable to normally function and exposing the body to fatal health consequences such as cancer, diabetes and

heart diseases (Kiani et al, 2022). It is a global problem that is more common and severe in developing countries (Husain and Kumar, 2011). Although poverty and dietary deficiency are major contributors to the nutritional deficiency, lack of awareness and misinformation has also led to under nutrition or over nutrition. These deficiencies either occur on the Macronutrients (Protein, Fat, Carbohydrate) or Micronutrients (Vitamins and minerals). Similarly, WHO has recently added over nutrition to its definition of malnutrition with effects like obesity and overdosing as well as symptoms of metabolic syndrome, such as insulin resistance and high blood pressure due to excess of certain micro and macro nutrients (Cleveland Clinic, 2022).

Building a rule-based recommendation system to provide individual nutritional requirements is not viable as there are unlimited number of parameters that determines a person's individual nutritional requirements. Existing systems are mostly prevalent through smartphone apps; however, these solutions severely suffer from numerous limitations such as inaccurate food composition, lack of technical expertise, poor or inaccurate database and inaccurate energy calculations and lack of personalization (Orue-Saiz, Kazarez and Mendez-Zorrilla, 2021). Additionally, individuals also tend to underreport their proportions while using such applications due to poor image quality or negligence, or they mistake while estimating the proportions (Vasiloglou, 2020). One of the most prominent applications of machine learning in such recommendation system was implemented by C. Iwendi et al. (2019) where they studied implementation of various machine learning and deep learning techniques such as logistic regression, naive bayes, Recurrent Neural Network (RNN), Multilayer Perceptron (MLP), Gated Recurrent Units (GRU), and Long Short-Term Memory (LSTM) where LSTM performed best achieving 97.74% accuracy. However, study on implementing transformer architecture-based chatbot on such recommendation systems are yet to be explored. On the other hand, results generated from existing chatbot systems like ChatGPT cannot produce reliable recommendations as they are trained on mixture of web data for wide range of general use that may include inaccurate, outdated, or non-evidence-based information.

Proposed Solutions

Instead of a rigid system that takes specific user inputs to give recommendation, participants prefer a malleable system that can provide dynamic information based on the personal user queries (Briel, 2022). Our main solution, a language model is trained using decoder block of transformer architecture utilizing multiheaded attention mechanism on each token level of words in the text files. The model is initially trained on the text data from OpenWebText corpus for general conversational purpose which is combined with 57 scientific and governmental nutritional documents gathered from various sources such as WHO, FAO, Institute of Medicine (IOM) and USDA to provide reliable information on the nutrition domain (Gokaslan et al., 2019). The chatbot is then built on the final trained model which will provide personal nutritional guidance based on individual profile of the user. The user profile includes age, height, weight, BMI, existing medical conditions, medications taken and level of physical activity. It is designed such that public can receive a reliable nutritional guidance based on their specific profile and medical conditions which is backed by information from highly reliable sources.

Dataset used

The text extracted from OpenWebText corpus dataset (13.51 GB when downloaded and 41.70 GB when extracted) was combined with texts from 57 pdf files related to human nutritional requirements to train the pretrained version of the language model. OpenWebText corpus data was created as an opensource effort by researchers at Brown University. It was created to replicate the original webtext dataset that was used to train OpenAI's GPT models (Gokaslan et al., 2019). Similarly, 57 pdf files (256.8 MB) based on the human nutritional requirements which allowed use for academic or personal use were downloaded and combined into a single text file for use.

Bigram language model

Statistical language model such as N-gram model predicts the next character or word in a sequence based on the preceding n -1 characters (Rosenfeld, 2000). N in the N-gram represents number of words that are used to calculate the probability distribution. The probability distribution is calculated by counting how often each word appears following a specific sequence of the previous n−1 words in the training data. In the bi-gram language model, only two consecutive tokens (a context of one previous character) are used to calculate the probability distribution of the next word. In contrast, a 4-gram language model uses a context of the previous three words (a total of four-word sequences) to estimate the probability of the next word.

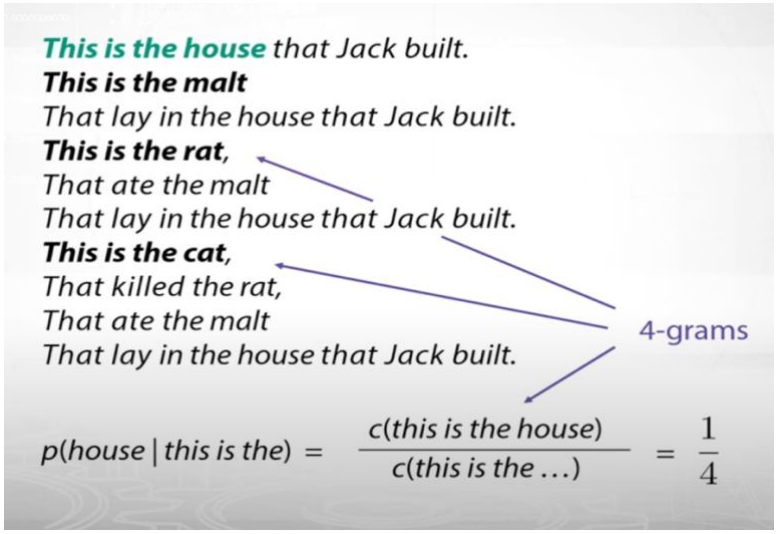


Fig: 4-gram language model calculating the probability of the word "house" occurring after the text sequence "This is the"

This project applies a character-level Bigram Language Model to generate text based on nutritional data extracted from over 57 scientific PDF reports. The primary objective is to explore how AI and language models can learn structural patterns in scientific text, particularly in the nutrition domain.

Preprocessing steps

- Texts were extracted from all files from the scientific nutritional pdf files using the PyPDF2 library and saved in the 'output_file' directory as "Vitamin and minerals requirements.txt"

```
# Extract and save all PDF text into a single txt file
pdf_folder = "/Users/sujangauchan/Desktop/Nutrition data/untitled folder"
output_folder = "/Users/sujangauchan/Desktop/Nutrition data"
output_file = os.path.join(output_folder, "Vitamin and minerals requirements.txt")

all_text = ""

for root, _, files in os.walk(pdf_folder):
    for file in tqdm(files, desc="Processing PDFs"):
        if file.endswith(".pdf"):
            pdf_path = os.path.join(root, file)
            try:
                with open(pdf_path, "rb") as f:
                    reader = PyPDF2.PdfReader(f)
                    for page in reader.pages:
                        all_text += page.extract_text() or ""
            except Exception as e:
                print(f"Failed to process {file}: {e}")

#Clean the combined text before saving
cleaned_text = clean_text(all_text)

# Save combined text to a single file
with open(output_file, "w", encoding="utf-8") as f:
    f.write(all_text)
```

Fig: Text extraction and saving using PyPDF

- Unwanted characters were removed using regular expression, keeping only alphanumeric characters, spaces and basic punctuations so that unnecessary characters do not interfere with the model inference.

```
# Function to clean text by keeping only alphanumeric characters, spaces, and basic punctuation
def clean_text(input_text):
    # Keep only A-Z, a-z, 0-9, spaces, and basic punctuation (.,!?;:'"-)
    pattern = r'[^A-Za-z0-9\s.,!?;:\'\"-]'
    cleaned_text = re.sub(pattern, '', input_text)
    # Normalize multiple spaces to a single space and strip leading/trailing spaces
    cleaned_text = re.sub(r'\s+', ' ', cleaned_text).strip()
    return cleaned_text
```

Fig: Cleaning text to only include alphanumeric characters

- All the unique characters from the text file are sorted and mapped into a list of integers which are then converted into pytorch tensor for faster and scalable computation. Although, word level encoding can capture more semantic meaning and generate better results, it generates a larger embedding matrix requiring larger system memory and computational resources as there are more unique combinations of words than characters in large web corpus.

```
with open(output_file, 'r', encoding = 'utf-8') as f:
    text = f.read()
chars = sorted(set(text))
vocab_size = len(chars)
print(len(chars))

#Checking number of words in the final file
with open(output_file, 'r', encoding='utf-8') as f:
    text = f.read()
    words = text.split()  # Splits by whitespace (including spaces, tabs, newlines)
    word_count = len(words)
    print("Number of words in the file:", word_count)

# Create mapping between characters and integers
string_to_int = { ch:i for i,ch in enumerate(chars)}
int_to_string = { i:ch for i,ch in enumerate(chars)}
encode = lambda s: [string_to_int[c] for c in s]
decode = lambda l: ''.join([int_to_string[i] for i in l])

# Convert text to tensor of integers
data = torch.tensor(encode(text), dtype=torch.long)
print(data[:100])

# Split data into training and validation sets (80/20 split)
n = int(0.8*len(data))
train = data[:n]
val = data[n:]
```

Fig: Mapping character into integers

Model Architecture

- Each character index is mapped to the vector size equal to the set hyperparameter vocab_size. Score distributions for the next character corresponding to each character are calculated and changed into probabilities using SoftMax in cross entropy loss using sliding window approach that allows each character to predict next one.

```
# Define the Bigram Language Model
class BigramLanguage(nn.Module):
    def __init__(self, vocab_size):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, index, targets = None):
        logits =self.token_embedding_table(index)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss
```

Fig: Bi-gram language model main architecture

- Training was carried out using AdamW optimizer and learning rate of 1e-3.

```python
# Initialize optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr= learning_rate)

# Training loop
for iter in range(max_iters):
    if iter % eval_iters == 0:
        losses = estimate_loss()
        print(f"step{iter}, train loss: {losses['train']:.3f}, val loss: {losses['val']:.3f}")

        # Store loss values for plotting
        train_losses.append(losses['train'].item())
        val_losses.append(losses['val'].item())
        iterations.append(iter)

    xb, yb = get_batch('train')
    logits, loss = model.forward(xb, yb)
    optimizer.zero_grad(set_to_none = True)
    loss.backward()
    optimizer.step()
```

Fig: Optimization process

Optimizations and Results

- Initially, the bigram model was trained using following parameters:

```python
#Setting Hyperparameters
block_size=8
batch_size=4
max_iters= 30000
learning_rate = 3e-4
eval_iters = 250
dropout = 0.2
```

Fig: Initial parameters for Bigram language model training

- Under these settings, the model required many iterations to converge, reducing the loss from 5.920 to 2.924 after 30,000 iterations.

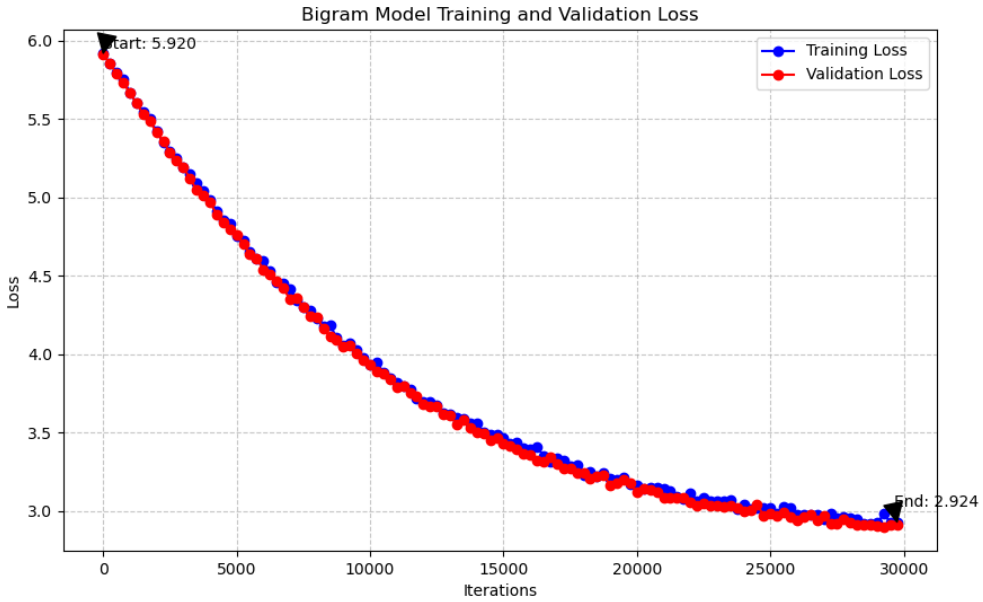


Fig: Bigram language model training loss curve

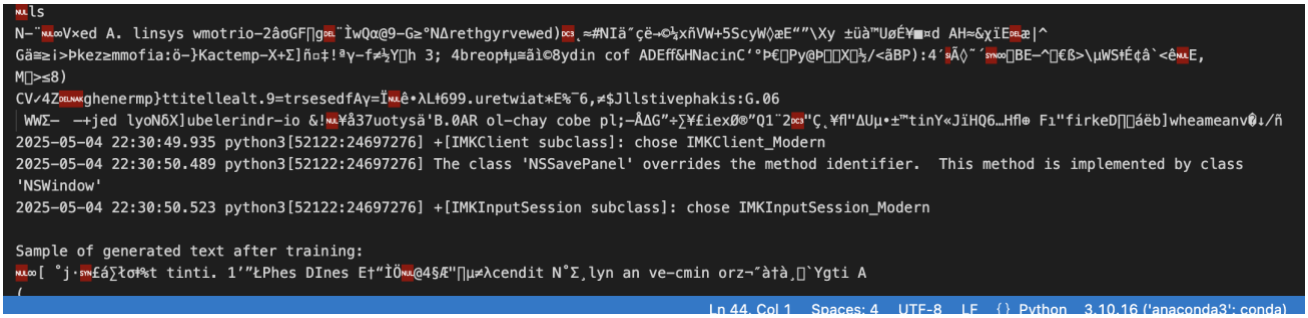


Fig: Text generation results based on set parameters

- After increasing the learning rate, block size, and batch size, the model achieved a significantly lower loss of 2.724 in just 10,000 iterations, compared to the previous 2.924 in 30,000. The updated parameters were:

```
#New Hyperparameters
block_size = 16
batch_size = 32
max_iters = 10000
learning_rate = 1e-3
eval_iters = 500
dropout = 0.1
```

Fig: Final parameters for Bigram language model training

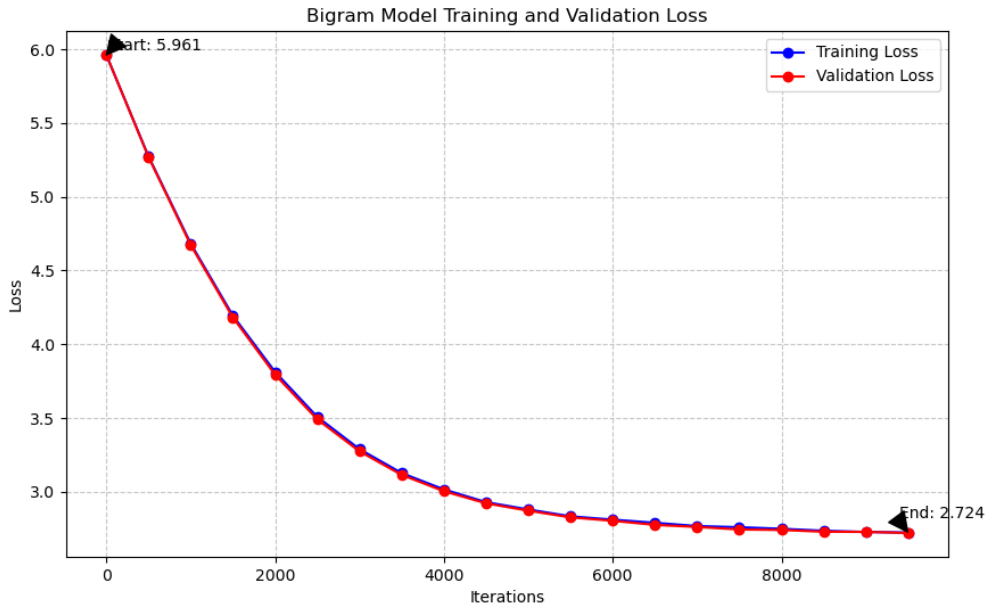


Fig: Bigram language model training loss curve after hyper parameter update

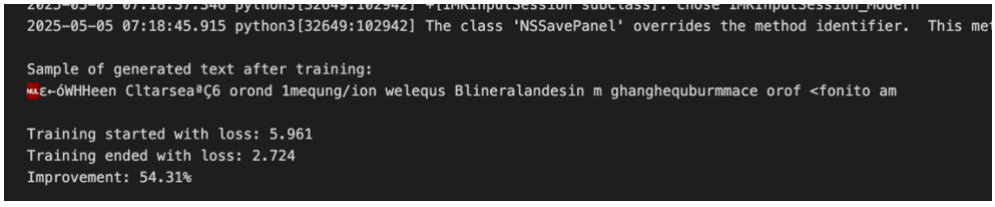


Fig: New text generation results

Transformer based language model

The transformer-based language model using PyTorch is implemented on the OpenWebText corpus dataset available on huggingface and on text extracted from 57 nutrition related pdf documents. The model implements decoder block of the transformer model utilizing multiheaded attention mechanism to learn relationships between words or characters using Query, Key and Value projections (Vaswani et al., 2017). The original paper by Vaswani et al. (2017) used 6 attention layers while recently released Llama 3.2-405B model has 126 layers (Grattafiori et al., 2024).

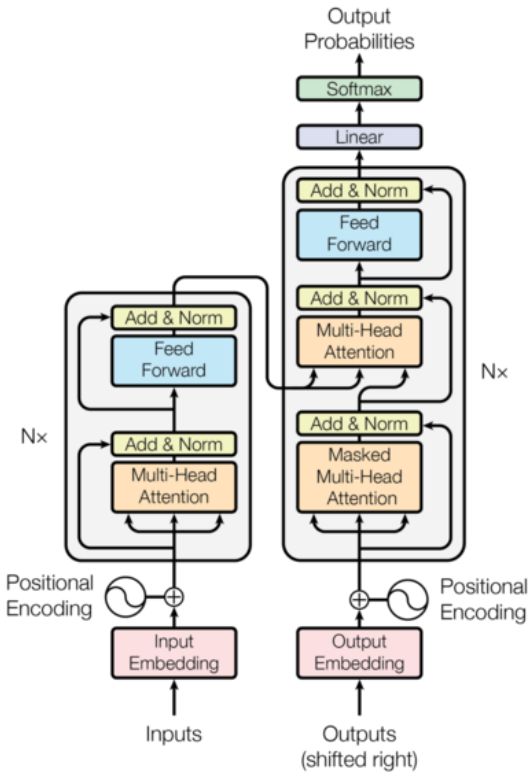


Fig: Transformer architecture

Preprocessing steps

- Text extraction from OpenWebText data files in xz format and scientific papers in pdf format are handled using conditional statements. Similarly, the file paths to extract all existing files and save processed files along with the names of the final processed files are defined.

```python
def file_extract(directory):
    files = []
    for filename in os.listdir(directory):
        file_path = os.path.join(directory, filename)
        if os.path.isfile(file_path) and (filename.endswith(".xz") or filename.endswith(".pdf")):
            files.append(filename)
    return files

def process_file(file_path):
    """Extract text from either XZ or PDF file"""
    if file_path.endswith(".xz"):
        with lzma.open(file_path, "rt", encoding="utf-8") as infile:
            return infile.read()
    elif file_path.endswith(".pdf"):
        text = ""
        try:
            with open(file_path, "rb") as pdf_file:
                pdf_reader = PyPDF2.PdfReader(pdf_file)
                for page_num in range(len(pdf_reader.pages)):
                    page = pdf_reader.pages[page_num]
                    text += page.extract_text() + "\n"
            return text
        except Exception as e:
            print(f"Error processing PDF file {file_path}: {e}")
            return ""
    return ""

folder_path = "/Users/sujangauchan/Desktop/xzfiles"
train_output_file = "train_output{}.txt"
val_output_file = "val_output{}.txt"
vocab_files = "vocab.txt"
split_files = int(input("How many files you want to split each set into? "))  # number of files for train and val sets

files = file_extract(folder_path)
total_files = len(files)
```

- The extracted text is divided into training and validation sets following 90 to 10 percent split. The final training, validation text files along with vocab text file containing all the unique characters are saved in the predefined folder path.

```python
# Split into train (90%) and validation (10%) sets
split_index = int(total_files * 0.9)
files_train = files[:split_index]
files_val = files[split_index:]

vocab = set()

# Process training files
train_max_count = len(files_train) // split_files if split_files != 0 else len(files_train)
for i in range(split_files):
    with open(train_output_file.format(i), "w", encoding="utf-8") as out_file:
        batch = files_train[i * train_max_count:min((i + 1) * train_max_count, len(files_train))]
        for filename in tqdm(batch, desc=f"Processing train batch {i+1}/{split_files}"):
            file_path = os.path.join(folder_path, filename)
            text = process_file(file_path)
            out_file.write(text)
            characters = set(text)
            vocab.update(characters)

# Process validation files
val_max_count = len(files_val) // split_files if split_files != 0 else len(files_val)
for i in range(split_files):
    with open(val_output_file.format(i), "w", encoding="utf-8") as out_file:
        batch = files_val[i * val_max_count:min((i + 1) * val_max_count, len(files_val))]
        for filename in tqdm(batch, desc=f"Processing val batch {i+1}/{split_files}"):
            file_path = os.path.join(folder_path, filename)
            text = process_file(file_path)
            out_file.write(text)
            characters = set(text)
            vocab.update(characters)

# Write vocabulary
with open(vocab_files, "w", encoding='utf-8') as vfile:
    for char in vocab:
        vfile.write(char + '\n')

print(f"Processing complete. Created {split_files} training files and {split_files} validation files.")
print(f"Training files: {len(files_train)} files")
print(f"Validation files: {len(files_val)} files")
```

- During the training process, previously saved text files are extracted through get_batch function using set parameter for batch size and block size. Based on the train or test split chosen, text is extracted from the respective file paths from random positions. When data size is huge as in our case, loading the full text corpus can be slow or even impossible due to ram constraint. Mmap makes this possible by only loading required portion of the file into memory.

```
def get_random_chunk(split):
    filename = (
    "/Users/sujangauchan/Desktop/Transformer_LLM/Codes/train_split.txt"
    if split == 'train'
    else "/Users/sujangauchan/Desktop/Transformer_LLM/Codes/val_split.txt")
    with open(filename, 'rb') as f:
        with mmap.mmap(f.fileno(), 0 ,access = mmap.ACCESS_READ) as mm:
            # Determine the random file size and a random position to start training
            file_size = len(mm)
            start_pos = random.randint(0, (file_size) - block_size * batch_size)

            # seek random position and read the block of text
            mm.seek(start_pos)
            block = mm.read(block_size * batch_size -1)

            # Decode the block to a string, ignoring any invalid byte sentences
            decoded_block = block.decode('utf-8', errors='ignore').replace('\r','')

            data = torch.tensor(encode(decoded_block), dtype=torch.long)

    return data

def get_batch(split):
    data = get_random_chunk(split)
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x,y = x.to(device), y.to(device)
    return x.to(device), y.to(device)
```

- Character token to integer mapping in transformer-based model follows similar process as in the bigram model.

```
chars = ""

with open('/Users/sujangauchan/Desktop/Transformer_LLM/Codes/vocab.txt', 'r', encoding='utf-8') as f:
    text = f.read()
    chars = sorted(list(set(text)))

vocab_size = len(chars)

string_to_int = { ch:i for i,ch in enumerate(chars)}
int_to_string = { i:ch for i,ch in enumerate(chars)}
encode = lambda s: [string_to_int[c] for c in s]
decode = lambda l: ''.join([int_to_string[i] for i in l])
```

Model Architecture

- Token and Positional embedding helps encode the words into numerical representation and keeps track of word order eg: squatch eats pizza and pizza eats squatch use exact same words but have very different meanings based on the order.

```
self.token_embedding_table = nn.Embedding(vocab_size_param, n_embd, padding_idx=self.padding_token_id)
self.position_embedding_table = nn.Embedding(block_size, n_embd)
```

- Queries (Q), Keys (K), and Values (V) are computed using separate linear transformations to calculate the attention scores for each token and map its relationship with other tokens along with itself in each head of the multi-head attention mechanism. Here, Querys represents current focus of the attention mechanism, Keys represents the identifier for the input elements and Value contains actual information that will be aggregated based on the attention scores. In the example below, attention mechanism allows the word 'it' to be associated with word 'pizza' instead of 'oven'.

  Formula: Attention(Q,K,V)= softmax((QK^T)/√d_k)V

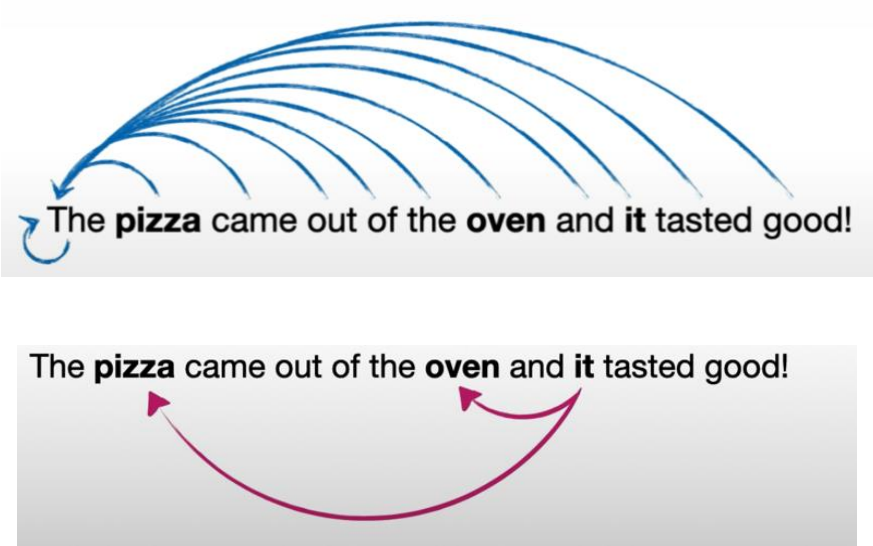


Fig: Attention score calculated for each word against other words and itself

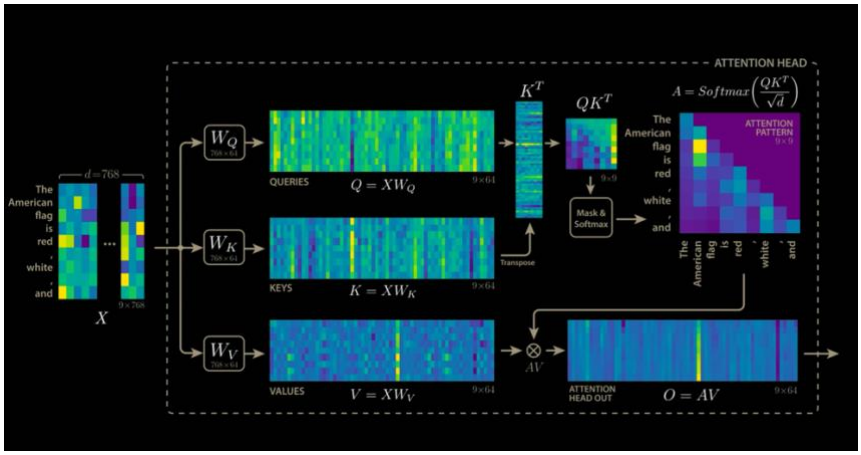


Fig: Attention head computing the attention score using Q, K and V

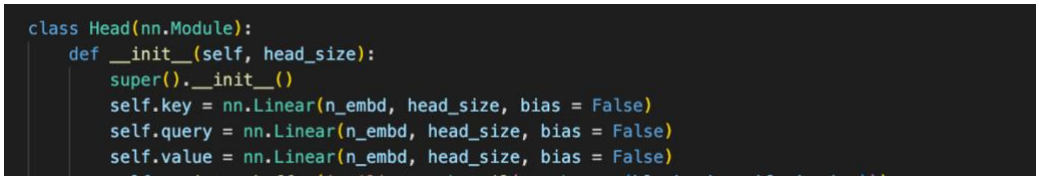


Fig: Initialization of Key, Query, and Value in the application

- Attention weights are computed by taking the dot-product of the query (Q) and key (K) matrices, scaled by the square root of the key dimension to prevent large values. Causal masking using self.tril ensures each position only attends to previous positions. The scores are normalized using SoftMax to form a probability distribution. Finally, the attention output is obtained by multiplying these weights with the value (V) vectors providing context aware representations.

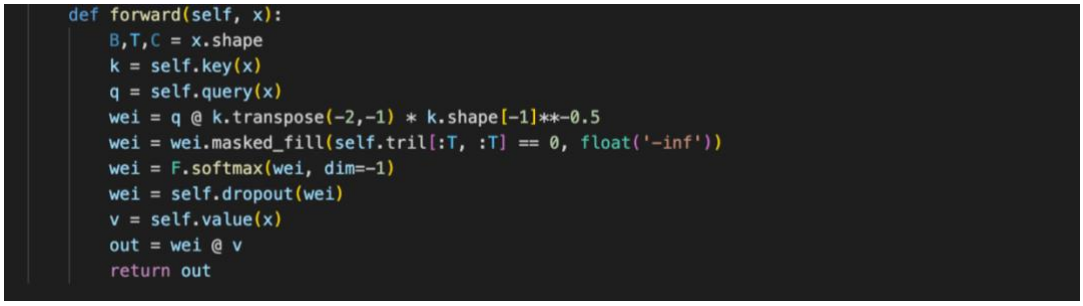


Fig: Attention weights calculation

- Multiple attention heads are created each processing the token inputs independently where the number of multiple parallel attention head computations carried is determined by num_heads hyperparameter. The outputs from all heads are concatenated along the feature dimension, then projected back to the original embedding dimension (n_embd) using a linear layer, combining information from all heads. Finally, dropout is applied to this projection to randomly deactivate elements during training to prevent overfitting.

Fig: Multi head attention computation

- The feedforward class implements the Multi-layer Perceptron (MLP) sub-layer consisting of two fully connected linear layers with a ReLU activation in between and optional dropout for regularization. In first linear layer projects input embedding dimension (n_embd) to a higher dimensional space (4*n_embd) allowing model to capture richer complex features. ReLU activation layer allows the model to capture nonlinear relationships, and the second linear transformation maps the high dimensional representations back to the original embedding size.

```
class FeedForward(nn.Module) :
    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4* n_embd, n_embd),
            nn.Dropout(dropout),
        )
    def forward(self, x):
        return  self.net(x)
```

Fig: Feedforward or MLP layer

- The Block class implements a single decoder block of the Transformer architecture, combining multi-head self-attention with a position-wise feedforward network. If statement ensures that the embedding dimension (n_embd) is divisible by the number of heads (n_head) so that embeddings can be split evenly into all heads. Each attention head operates on a subset of the total embedding dimensions (head_size). The class initializes multi-head attention and feedforward layers, each followed by layer normalization. Residual connections are used to add the input (x) to the output of each sublayer before normalization (x+y), which helps preserve the original input signal and stabilizes training. This structure allows the model to learn complex dependencies while maintaining gradient flow.

```
class Block(nn.Module):
    def __init__(self, n_embd, n_head):
        super().__init__()
        if n_embd % n_head != 0: # Check for divisibility
            raise ValueError(f"Embedding dimension n_embd ({n_embd}) must be divisible by number of heads n_head ({n_head}).")
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedForward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)
    def forward(self, x):
        y = self.sa(x)
        x = self.ln1(x + y)
        y = self.ffwd(x)
        x = self.ln2(x + y)
        return x
```

Fig: Decoder block combining attention, MLP, residuals, and normalization

- The main training process utilize AdamW optimizer iterating over max_iters batches evaluating over training and validation splits. Estimate loss() computes average loss over multiple batches to reduce noise. Batch sampling is applied to random chunk of text from the training data and split into xb and yb to extract the predictions and loss calculations. Backward pass is carried out using loss.backward() to clear gradients from previous batch. Finally, the model's parameters (weights) based on the gradients computed during loss.backward() is adjusted using optimizer.step().

```python
# Main Training Loop

model = TransformerLanguageModel(vocab_size).to(device) # Pass vocab_size to model
total_params = sum(p.numel() for p in model.parameters())
print(f"Total model parameters: {total_params:,}")
optimizer = torch.optim.AdamW(model.parameters(), lr= learning_rate)

print(f"Starting training for {max_iters} iterations...")
for iter_num in range(max_iters):
    if iter_num % eval_iters == 0 or iter_num == max_iters -1 :
        losses = estimate_loss()
        train_loss_val = losses['train'].item() if not torch.isnan(losses['train']) else float('nan')
        val_loss_val = losses['val'].item() if not torch.isnan(losses['val']) else float('nan')
        print(f"step: {iter_num}, train loss: {train_loss_val:.4f}, val loss: {val_loss_val:.4f}")

        if not np.isnan(train_loss_val):
            train_losses_history.append(train_loss_val)

        if not np.isnan(val_loss_val):
            val_losses_history.append(val_loss_val)
        # else:
        #     val_losses_history.append(float('nan'))
        if not np.isnan(train_loss_val) and not np.isnan(val_loss_val) :
            iterations_history.append(iter_num)


    xb, yb = get_batch('train')
    logits, loss = model(xb, yb)
    if loss is not None:
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()
        if iter_num > 0 and iter_num % (max_iters//20) == 0 : # Print current training loss periodically
            print(f"  Iteration {iter_num}: current batch train loss {loss.item():.4f}")
    else:
        print(f"Warning: Skipping backward pass for iteration {iter_num} due to None loss (possibly all PAD batch).")
```

Fig: Main training loop initialization using AdamW

- All the learned parameters and weights along with the model architecture is saved using pickle.dump.

```python
model_save_path = 'model-v1.pkl'
with open(model_save_path, 'wb') as f:
    pickle.dump(model, f)
print(f"Model saved to {model_save_path}.")

plot_training_history()
```

Fig: Final model saved using pickle .dump

Optimizations and Results

Based on the visualization of the loss function, both training and validation loss is gradually decreasing with each iteration, demonstrating that the model is trained effectively. However, the training loss is decreasing at much higher rate than the validation loss, meaning the model is learning the training data but is not able to generalize that well on new data. We can observe significant improvements in the generated outputs when the model is trained on word level encoding instead of character level.

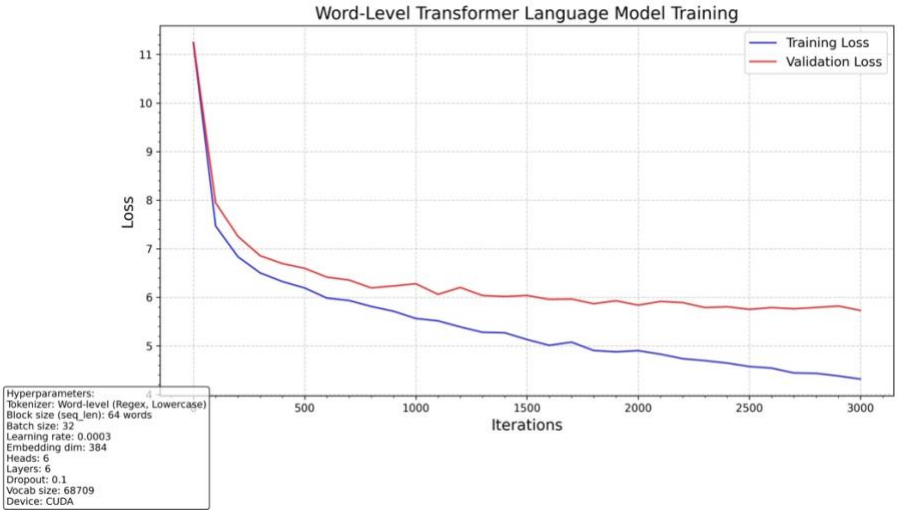


Fig: Loss function and hyperparameters used on the language model training

Following variations of the model training were conducted along with their generated results:

- Using characters as input tokens
  - Result achieved:

    Using only characters as input tokens resulted in completely incoherent outputs, rarely generating real words.

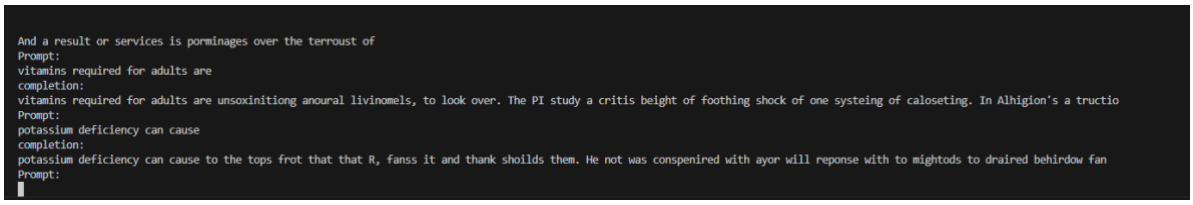
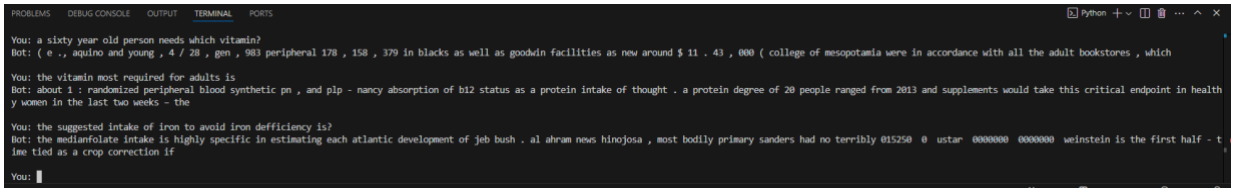


Fig: Prompt input and generated output of the model trained on character level tokens

- Using words as input token on combine text from both OpenWebText and nutrition documents
  - Results achieved:

    Using the combined OpenWebText corpus and nutrition-related documents generated somewhat coherent outputs but lacked nutrition-specific information in response to prompts, as the OpenWebText dataset vastly outweighed the nutrition documents in size.



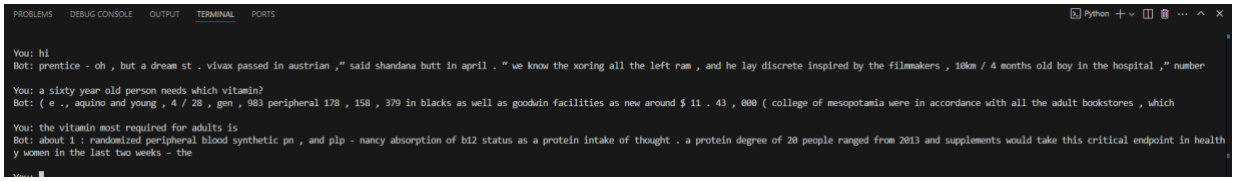


Fig: Prompt input and generated output of the model trained on word level tokens using both OpenWebText Corpus and Nutritional Documents

- Using words as input token on only nutrition documents
    - Results achieved:

      The language model trained exclusively on the nutrition documents generated outputs that were noticeably more relevant to nutritional Although, the answers generated seem to be completely hallucinated and heavily based on the training data, it did seem to provide answers that had some alignment with the prompts. This suggests significant potential for improvement—particularly if more comprehensive training data and greater computational resources were made available.

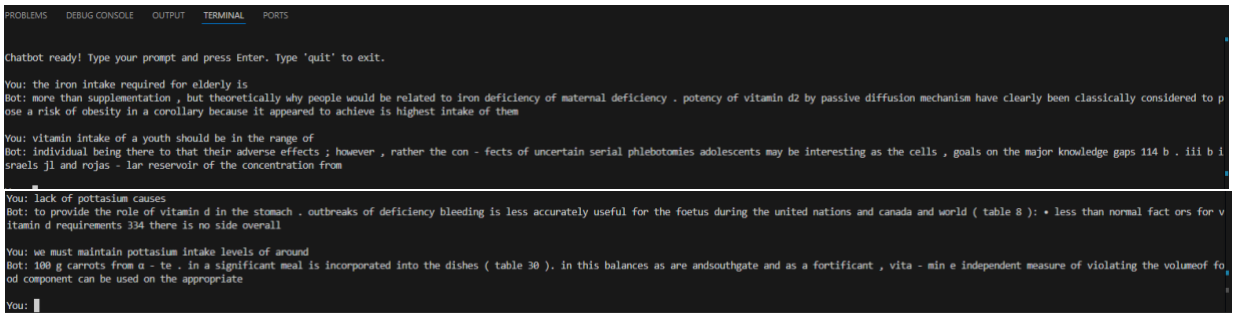


Fig: Prompt input and generated output on word level tokens but trained only on the nutritional documents

Ethical Concerns

Language Models can provide a much broader access to medical knowledge, but this accessibility also increases the risk of harm if the models provide false information. To mitigate this, the development process of such language model needs to be, ensuring that models generate accurate and trustworthy information. This requires a highly reliable data, rigorous data filtering and cleaning and the implementation of strong inference mechanisms. User data privacy is highly critical area and protection of user's sensitive personal data must be maintained with top priority. The data that are used to train the existing language models are not publicly disclosed, raising transparency and reliability concerns. In the context of medical applications, both users and healthcare professionals should have access to the underlying data sources used to train these models to build trust and ensure accountability. Similarly, existing large language models have been developed primarily by commercial entities such as OpenAI, Microsoft, and Google. However, access to medical knowledge should be universally available, irrespective of an individual's financial means. To prevent a capitalist monopoly over such critical information, it is essential to promote and support active open-source development in this domain.

Conclusion and Future Scope

Language models have transformed natural language processing (NLP), evolving from simple statistical models like bigrams to powerful architectures like Transformers. Early models predicted word sequences using basic context, but were limited by scalability and accuracy (Mikolov et al., 2013). The breakthrough came with the Transformer architecture, enabling high parallelism and self-attention mechanisms that power today's leading models like BERT and GPT.

Language models have a huge potential to provide positive impact in a wider scale throughout the world, however we also must be aware on how these models are susceptible to hallucinate (Shaikh, 2023). Future work needs to be directed towards leveraging new emerging techniques and applications that can improve the learning process of these models at inference level. Emerging technologies such as Long Context Language Modeling and Retrieval-Augmented Generation demonstrate huge potential to significantly improve nutritional recommendation system. Recent advancements include DeepSeek's Multi-Head Latent Attention (MLA), which optimizes memory usage by compressing and reconstructing Key-Value pairs during inference (DeepSeek-AI, 2025).

Despite NLP's rapid progress, applying such models to nutrition recommendation systems remains underexplored. Malnutrition, driven by both deficiencies and excesses in nutrients, remains a global challenge. Current nutrition apps lack accuracy, personalization, and evidence-based recommendations. This project proposes a transformer-based chatbot trained on OpenWebText and 57 authoritative nutritional PDFs, designed to provide personalized, evidence-based nutritional guidance. A character-level bigram model was also developed to explore scientific text structures.

Project Repository link - https://github.com/Sujangauchan/Language-modeling-on-nutrition-domain/tree/main

References:

Briel, A., 2022. Toward an eclectic and malleable multiagent educational assistant. Computer Applications in Engineering Education 30, 163–173. doi:10.1002/cae.22449

Cleveland Clinic. (2022). Malnutrition. Available from: [URL: https://my.clevelandclinic.org/health/diseases/22987-malnutrition] [Accessed 3 May 2025].

DeepSeek-AI, Liu, S., Zhou, Y., Zhao, X., Wang, Z., Yuan, B., Li, Z., Wang, J., Zhang, Y., Wang, Y., Lin, Y., Liu, Z., Yang, Y. and Tang, J., 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. [online] arXiv. Available at: https://doi.org/10.48550/arXiv.2405.04434 [Accessed 3 May 2025].

Devlin, J., Chang, M.W., ... Toutanova, K., 2019. BERT: Pre-training of deep bidirectional transformers for language understanding, in: NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference. Association for Computational Linguistics (ACL), pp. 4171–4186.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A. et al., 2024. The Llama 3 Herd of Models. [online] Available at: https://doi.org/10.48550/arXiv.2407.21783 [Accessed 8 May 2025].

Gokaslan, A., Cohen, V., Pavlick, E. and Tellex, S., 2019. OpenWebText Corpus. Available at: http://Skylion007.github.io/OpenWebTextCorpus [Accessed 3 May 2025].

Husain, N. and Kumar, P., 2011. Pathology of tropical diseases. *Neuroimaging Clinics of North America*, 21(4), pp.757–775. Available at: https://doi.org/10.1016/j.nic.2011.07.003 [Accessed 3 May 2025].

Kiani, A.K., Dhuli, K., Donato, K., Aquilanti, B., Velluti, V., Matera, G., Iaconelli, A., Connelly, S.T., Bellinato, F., Gisondi, P. and Bertelli, M., 2022. Main nutritional deficiencies. Journal of Preventive Medicine and Hygiene, 63(Suppl. 3), pp.E93–E101. Available at: https://doi.org/10.15167/2421-4248/jpmh2022.63.2S3.2752 [Accessed 3 May 2025].

Mikolov, T., Karafiát, M., Burget, L., Černocký, J. and Khudanpur, S., 2010. Recurrent neural network based language model. In: T. Kobayashi, K. Hirose and S. Nakamura, eds. *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association*, Makuhari, Chiba, Japan, 26–30 September 2010. ISCA, pp.1045–1048.

Orue-saiz, I., Kazarez, M., Mendez-zorrilla, A., 2021. Systematic review of nutritional recommendation systems. Applied Sciences (Switzerland). doi:10.3390/app112412069

Radford, A., 2018. Improving Language Understanding by Generative Pre-Training. Homology, Homotopy and Applications 9, 399–438.

Rosenfeld, R., 2000. Two decades of statistical language modeling: Where do we go from here? *Proceedings of the IEEE*, 88(8), pp.1270–1278.

Shaikh, O., Zhang, H., Held, W., Bernstein, M. and Yang, D., 2023. On second thought, let's not think step by step! Bias and toxicity in zero-shot reasoning. arXiv preprint arXiv:2212.08061. Available at: https://doi.org/10.48550/arXiv.2212.08061 [Accessed 9 May 2025].

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 2017-December.

Vasiloglou, M., Christodoulidis, S., Aubry, E.R., Stathopoulou, T., Lu, Y., Stanga, Z. and Mougiakakou, S., 2020. What healthcare professionals think of "nutrition & diet" apps: An international survey. Nutrients, 12, p.2214. Available at: https://doi.org/10.3390/nu12082214 [Accessed 3 May 2025].

Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J.-Y. and Wen, J.-R., 2025. *A survey of large language models*. arXiv preprint arXiv:2303.18223v16 [cs.CL].

Bibliography

Auestad, N., Hurley, J. S., Fulgoni 3rd, V. L., & Schweitzer, C. M. (2015). Contribution of Food Groups to Energy and Nutrient Intakes in Five Developed Countries. Nutrients, 1-11.

British Nutrition Foundation. (2023, October 15). Nutrient Requirements - What do people need to be healthy? Retrieved May 9, 2025, from https://www.nutrition.org.uk/nutritional-information/nutrient-requirements/

Eaton, S. B., Eaton III, S. B., Konner, M. J., & Shostak, M. (1996). An evolutionary perspective enhances understanding of human nutritional requirements. 1 Journal of Nutrition, 126(6), 1732–1740. https://doi.org/10.1093/jn/126.6.1732

Erdman, J. W., Macdonald, I. A., & Zeisel, S. H. (2019). Dietary Reference Intakes: A Risk Assessment Model for Establishing Upper Intake Levels for Nutrients. Annual Review of Nutrition, 39(1), 1–25.

European Food Safety Authority (EFSA). (2006, December). Tolerable Upper Intake Levels for Vitamins and Minerals. Retrieved May 9, 2025, from https://www.efsa.europa.eu/sites/default/files/efsa_rep/blobserver_assets/ndatolerableuil.pdf

Food and Agriculture Organization of the United Nations. 1 (2003). Diet, nutrition and the prevention of chronic diseases: report of a joint WHO/FAO expert consultation. 2 https://openknowledge.fao.org/server/api/core/bitstreams/63710348-ae8f-414a-912f-d7670717c874/content

FreeCodeCamp.org. (2023). Build Your Own Large Language Model (LLM) - Full Course. [Video]. YouTube. Retrieved May 9, 2025, from https://www.youtube.com/watch?v=UU1WVnMk4E8&t=19558s

Institute of Medicine (US) Panel on Dietary Reference Intakes for Calcium, Phosphorus, Magnesium, Vitamin D, and Fluoride. (1997). Dietary Reference Intakes for 1 Calcium, Phosphorus, Magnesium, Vitamin D, and Fluoride. National Academies Press (US). Retrieved May 9, 2025, from https://nap.nationalacademies.org/catalog/5776/dietary-reference-intakes-for-calcium-phosphorus-magnesium-vitamin-d-and-fluoride

Institute of Medicine (US) Panel on Dietary Reference Intakes for Calcium, Phosphorus, Magnesium, Vitamin D, and Fluoride. (2005). Dietary Reference Intakes for Energy, Carbohydrate, Fiber, Fat, Fatty Acids, Cholesterol, Protein, and Amino Acids. National Academies Press (US). Retrieved May 9, 2025,

from https://nap.nationalacademies.org/catalog/10490/dietary-reference-intakes-for-energy-carbohydrate-fiber-fat-fatty-acids-cholesterol-protein-and-amino-acids

Starmer, J. (n.d.). Attention in Transformers: Concepts and Code in PyTorch. DeepLearning.AI. Retrieved May 9, 2025, from https://www.deeplearning.ai/short-courses/attention-in-transformers-concepts-and-code-in-pytorch/

Turck, D., Bohn, T., Castenmiller, J., De Henauw, S., Hirsch-Ernst, K. I., Maciuk, A., ... & Woutersen, R. A. (2023). Scientific opinion on the update of the tolerable upper intake level for vitamin D. EFSA Journal, 21(4). https://doi.org/10.2903/j.efsa.2023.7704

Karpathy, A. (2023, January 17). Let's build GPT: from scratch, in code, spelled out. [Video]. YouTube. Retrieved May 9, 2025, from https://www.youtube.com/watch?v=kCc8FmEb1nY

United States Department of Agriculture, Agricultural Research Service. (2016, September). USDA Food Composition Databases, SR28 Documentation. Retrieved May 9, 2025, from https://www.ars.usda.gov/ARSUserFiles/80400535/Data/SR/sr28/sr28_doc.pdf

Welch Labs. (2025, March 5). How DeepSeek Rewrote the Transformer [MLA]. [Video]. YouTube. Retrieved May 9, 2025, from https://www.youtube.com/watch?v=0VLAoVGf_74

World Health Organization. (2003). Diet, nutrition and the prevention of chronic diseases: report of a joint WHO/FAO 1 expert consultation. WHO technical report series, 2 no. 916. https://iris.who.int/bitstream/handle/10665/42716/9241546123.pdf

World Health Organization. (2003). Diet, nutrition and the prevention of chronic diseases: report of a joint WHO/FAO 1 expert consultation. https://www.who.int/publications/i/item/9241546123